Computer System Support for Scientific and Engineering Computation

Lecture 8 - May 26, 1988 (notes revised June 13, 1990)

1 A Peek at the IEEE Standard

Before we discuss several features of the IEEE standard for floating-point arithmetic, the following two misprints in the document ANSI/IEEE Std 854-1987, *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, should be noted :

1. On the right hand column of page 8, replace

   (i) Numbers for the form $(-1)^s b^E (d_0 d_1 d_2 \cdots d_{p-1}), \ldots$

   by

   (i) Numbers for the form $(-1)^s b^E (d_0 \,.\, d_1 d_2 \cdots d_{p-1}), \ldots$

2. At the bottom of the left hand column of page 16, replace

   For x positive and finite, ...

   by

   For x *nonzero* and finite, ...

1.1 Radices

The IEEE standards do not permit as many different radices as they appear to allow. Even though the IEEE standard 854 is "Radix-Independent", radices 2 and 10 are the only two supported; IEEE standard 754 allows only radix 2. The restriction on the radix does not appear anywhere else in the text of the 854 standard aside from the statement indicating the restriction. The reason for the limitation is mainly political more than anything else. There is no way for IBM or its imitators, all of whom use hexadecimal as radix, to conform to the rounding schemes of the IEEE standard without massive changes to current code. For the same reason it is inconceivable for Burroughs, which uses octal as radix, to conform to the standard. As a consequence, it serves no purpose to support octal, hexadecimal or other radices beside binary or decimal, since it is unlikely that IBM or Burroughs will redesign their old products to conform to any IEEE standard. The IEEE standards are,

therefore, applicable to new machines designed and built in recent years, most of which use binary or decimal radices.

The reasons for preferring binary radix, $\beta = 2$, to decimal radix, $\beta = 10$, or vice-versa, are very weak. Decimal radix is most appropriate for people who actually look at the output. The HP 71B, which uses radix 10, conforms to the IEEE 854 standard almost perfectly. Unfortunately, there are not many machines which conform to the standard and use radix 10, so there is not much software for these machines. Until there are more of them there will not be much software designed for them, and until there is sufficient software support people are reluctant to build these machines. This vicious cycle is unlikely to be broken.

Lots of machines conform to the IEEE standard 754 using radix 2. The reason for preferring $\beta = 2$ is not primarily because people don't want to see the output, but because binary has many advantages: simpler error analysis, for instance.

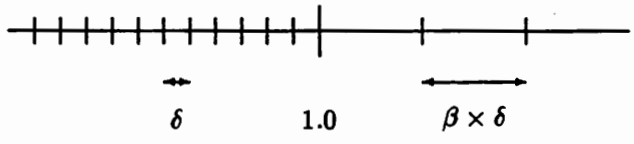
Consider the following statement in a language like Fortran :

$$X = B \,\square\, C,$$

where $\square$ is a floating point operation like $+, -, \times$ or $/$. Because of the way numbers are stored in memory, we actually find that

$$x = (b \,\square\, c)(1 \pm \xi),$$

where x, b, c are the respective values stored for the variables whose names are X, B, C and $\pm\xi$ is a small quantity introduced by roundoff. The trouble here is that the small perturbation ξ is not bounded sharply. Because of the way floating point works, the bound ξ is pessimistic for some numbers: pessimistic by a factor β, the radix used. When we consider representable numbers less than and greater than 1.0, we see that the gaps for the numbers greater than 1.0 are larger than those smaller than 1.0. In fact, the gaps between representable numbers greater than 1.0 are β times larger, as shown below :



Since the uncertainty ξ is a bound for all possible errors, it must be able to accommodate errors arising from representing numbers both to the left and right of 1.0. Therefore, the uncertainty ξ is especially pessimistic for results x just smaller than a power of the radix when a large radix is used, which makes error analysis somewhat ugly.

There is a peculiar propensity among micro-optimizing programmers to optimize certain kind of expressions. For example, consider the following :

constant $C = 1.003215\ldots$

$\vdots$

$E = X \times C$

A conscientious programmer may replace the above code by

$$\begin{aligned} &\text{constant } D = 0.996795\ldots \\ &\quad\vdots \\ &E = X/D \end{aligned}$$

where the constant D is the reciprocal of the constant C. The reason for using D instead of C is because the expected magnitude of a rounding error for C is β times that for D. The fact is that until we actually consider the values of C and D closely, we don't really know the exact rounding error in each case. There are lots of examples where C is a better approximation than D. However, since we usually do not know the actual digits of C and D or what machines the program is supposed to run on, we may just divide X by D to be on the safe side.

Tricks such as those described above are actually used in practice. For instance, in the implementation of certain transcendental functions in the library, constants which suffer from less rounding error are used except when divisions are too much slower than multiplications.

Another way to compute E, more accurate than the previous two schemes, is by

$$E = X + \tilde{C}X$$

where $\tilde{C} = C - 1$. Since C is very close to 1.0, $\tilde{C}$ is quite small. As a result the rounding error in representing $\tilde{C}$ and in the multiplication are inconsequential, so effectively there is just one rounding error, that is, in the addition. The trick here is to throw in an extra arithmetic operation, the addition; in most cases, the total cost of an addition and a multiplication is less than a division.

1.2 Why Use Large Radices?

1.2.1 Effect of Radix on Speed

Why did IBM use hexadecimal radix? IBM's decision to use hexadecimal as its radix is based on some experiments performed by Sweeney. He discovered that, on binary machines, the shifts resulting from normalizations are usually short, often less than 4 bits. In other words, results obtained from arithmetic operations seldom require long normalization shifts of 4 bits or more. There is a proponderence of short shifts because shifts arising from multiplications and divisions are at most 1 bit, and even additions and subtractions rarely encounter massive cancellations.

Sweeney reasoned that if we forego short normalizations, computations can go considerably faster. His reasoning led to the idea of using hexadecimal as the radix in floating-point number representation.

Consider the way floating point-numbers are represented in the IBM 790 :

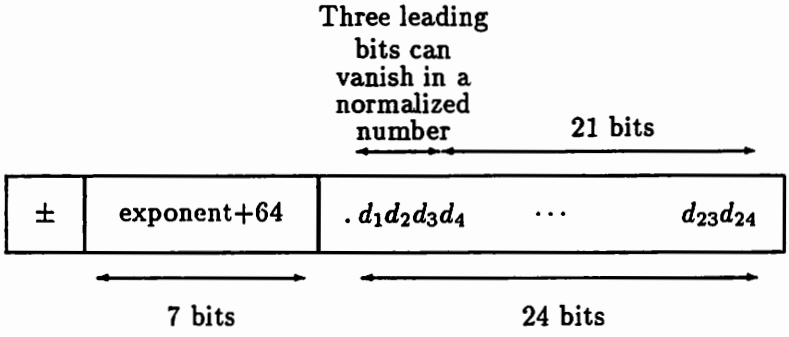
$\pm$	exponent+BIAS	$.d_1 d_2$	$\cdots$ $d_{p-1} d_p$

If $d_1 = 0$ normalizations are needed when the radix $\beta = 2$. Sweeney's tests indicate that it is quite rare for $d_j = 0$ where $j = 1, \ldots, k$ and $k \geq 4$. If a hexadecimal radix is used, no normalization is needed when $k \leq 3$. Thus, Sweeney's results suggest that we seldom need to normalize if hexadecimal radix is used, which means that much time can be saved.

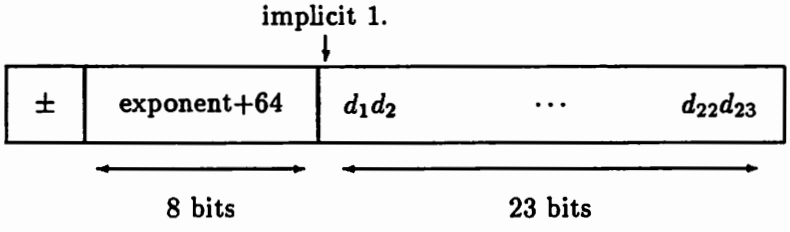
Actually there is another reason for using hexadecimal radix besides the reduction in the incidence of normalizations. Computers like the IBM 360 Model 30 had 1 byte wide busses. Because memory is accessed in bytes, it appeared that one could microcode arithmetic to run faster if one does not use single bit shifts, but uses nibble (half-byte) shifts instead. However, it turns out that the running time was lengthened for machines such as the the IBM 360 Model 65 and 75 because they had hardware for arbitrary binary shifts and were slightly slowed when restricted to nibble shifts.

1.2.2 Effect of Radix on Precision

The floating point representation for single precision numbers on an IBM machine is shown below :



whose range is from $\pm 2^{-260} = \pm 16^{-65}$ to $\pm 2^{252} = \pm 16^{63}$. The IEEE standard's single precision numbers are represented as :



whose range is from $\pm 2^{-128}$ to $\pm 2^{128}$.

For the IBM computers, a normalized number may have zeroes for the first three bits of its significand field, so in the worst case there are just 21 bits of precision. The IEEE standard, on the other hand, has 24 bits of precision (23 bits in the significand and 1 implicit bit). Thus, although the range for the IBM machines is larger than that of the IEEE standard's, the latter has 3 extra bits of precision.

On the IBM 7094, which had 27 bits in the significand field and about the same range

as IEEE single, there were not many complaints about underflow and overflow problems. The drop from 27 to 21 bits of precision was a major disaster. When 21 bits were used, the effect of roundoff became very apparent; programs which worked fine when 27 bits were used suddenly gave strangely inaccurate results.

This drop in the number of bits crossed the magic threshold where a rule of thumb states that one should carry at least about twice as many bits in the intermediate values as in the accuracy desired (see the discussion on *A Rule of Thumb for Working Precision* in lecture 2). If the number of digits used is more than the threshold, the chance is high that we will not see the effects of roundoff in the computations except in numerically unstable algorithms; in such algorithms, the best solution to circumvent problems due to roundoff error is to use a more stable algorithm, if available. Engineers often require about 12 bits of accuracy, so 27 bits are above the threshold whereas 21 bits are below. As a result of this phenomenon, single precision numbers with hexadecimal radix acquired a very bad reputation.

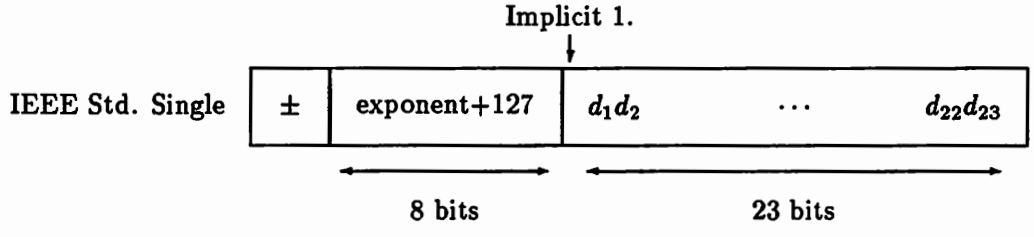
So the attempt to speed up floating-point arithmetic on a range of machines, by using hexadecimal radix, actually slowed down faster models. Furthermore, there is no evidence indicating that Sweeney performed his experiments on hexadecimal machines. If Sweeney's experiments were performed just on binary machines, his results may not apply to hexadecimal machines.

For reasons stated above the IEEE 854 standard does not include hexadecimal radices.

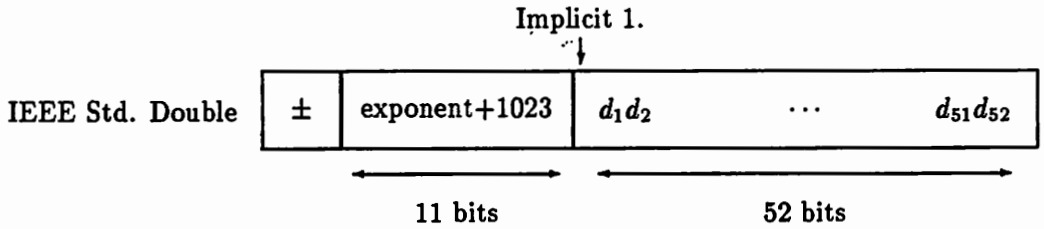
1.3 Single and Double Precisions

1.3.1 Exponent and Significand Widths

First consider the IEEE 754 standard floating point formats. The single precision format has 32 bits as depicted below :

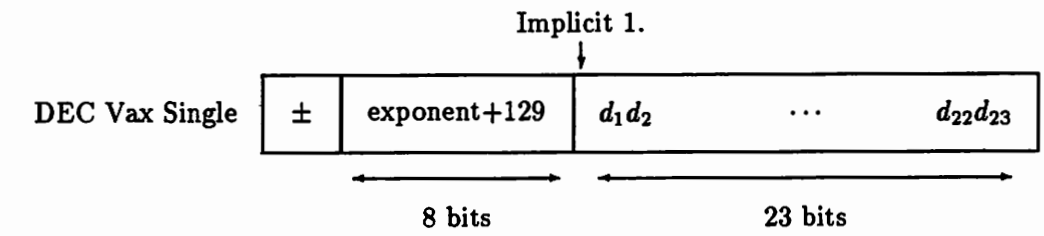


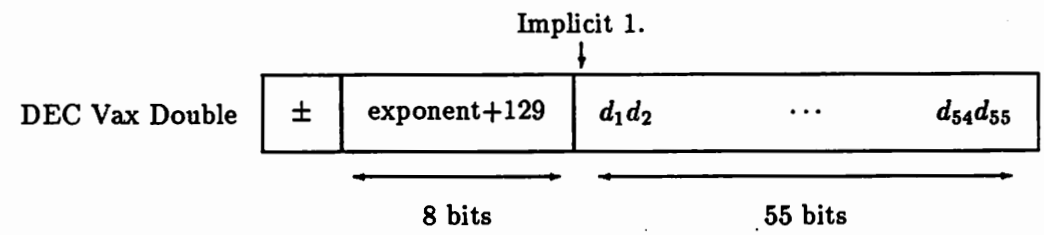
and the double precision has 64 bits as shown below :



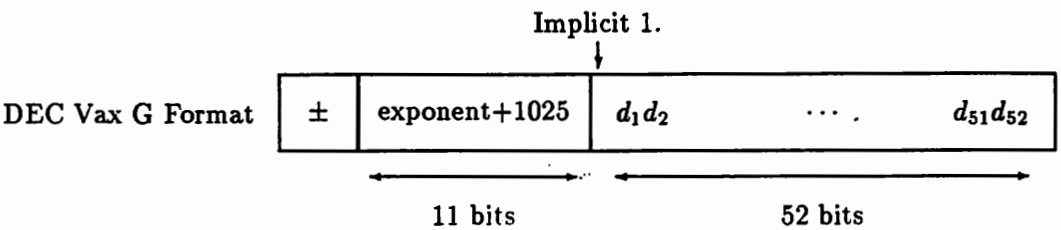
We will discuss another format in the IEEE standard, the extended format, in future lectures. The exponent widths of the single and double formats are different, 8 and 11 bits wide, respectively.

Consider another machine, the DEC Vax, whose floating point formats are quite similar to those of the IEEE standard. Its single and double formats are 32 and 64 bits wide, respectively, and are illustrated below :





The single and double formats are called "F" and "D" formats, respectively. There were rumors that DEC received numerous complaints about the D format because those of its customers who had come from using CDC Cybers too often encountered severe overflow and underflow problems. DEC introduced a new format, first in software then in microcode, to solve this problem. This new format, called the "G" format, is shown below :



It has 11 bits in the exponent field. Note that the DEC Vax F and G floating point formats are quite similar to those of the IEEE standard. However, in all of DEC's Vax formats, when

exponent+BIAS=0, the rest of the significant bits d_j are ignored: the number is treated as zero if the sign is "+", as a "reserved" operand if the sign is "-". The latter precipitates a trap when used as an operand for any floating point operation, even a MOVE.

What is actually wrong with the D format? Its exponent range is too narrow for its precision. The D format has 55 bits of precision, which means that the roundoff error, ξ, is of the order 2^{-55}. Unfortunately, the smallest representable number on this machine is just 2^{-128}. This means that if we cube the roundoff error, it will underflow as a consequence. Taking cubes of numbers are not uncommon; it arises quite often in many interesting matrix algorithms, such as the QR algorithm for solving eigenproblems. This means that on the DEC Vax D format, underflow may occur for certain algorithms, and the program will terminate prematurely. Ideally, we would like to be able to raise the rounding errors to at least the fourth power without any underflow problems. Notice that this requirement is satisfied by the F and G formats.

1.3.2 Portability of Programs

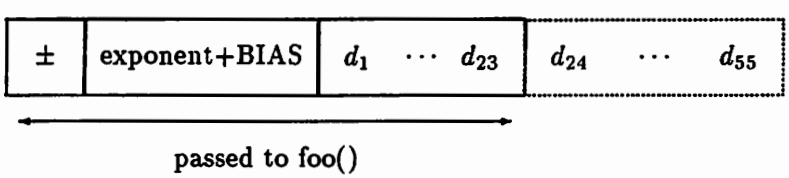
There is another kind of trouble which one has to deal with when porting programs between various computers. This problem has nothing to do with what was discussed previously, but arises because of the peculiarity of some computer languages that do not defend programs from mistyped arguments. For instance, assume that we have a function :

```
function foo (A,...)
single A
{
   ⋮
}
```

and in the main program, we have :

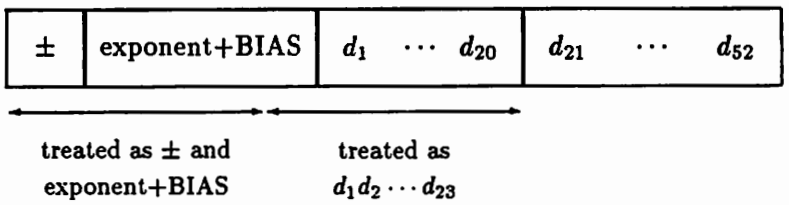
```
double D
   ⋮
··· foo (D,...)
   ⋮
```

Here, we have a situation where an argument declared as of double format is passed to a function which expects the argument to be of the single format. On a DEC Vax what happens appears to be pretty innocuous because the single precision number which foo() expects is within a rounding error of the first 4 bytes of the 8 byte double precision number as shown below :

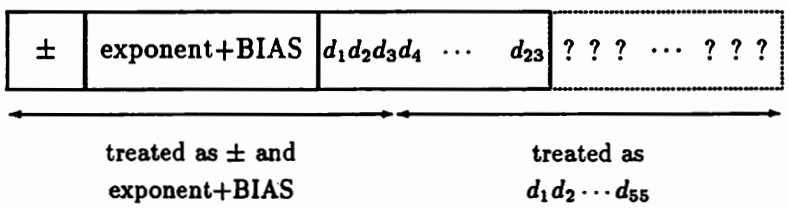


The value which is manipulated by foo() is a single precision entity which is the value of the double precision number chopped to a single precision number. This is the reason foo() produces reasonable answers.

On the other hand, if we had used machines with the IEEE format, we would not be so (un)lucky. On these machines, the exponent field of a single precision number is narrower than that of the double precision number. If foo() picks up a double precision number in lieu of a single precision number, the number it obtains most probably is not at all close to the number the caller intended. This situation is not unexpected since some of the exponent bits are treated as part of the significand field shown below :



The reverse situation, where a single precision number is passed to a function doo() which expects a double precision number, is even more interesting. Here we have a situation where doo() attaches arbitrary bits to the single precision number passed to it and treats them as part of the double precision number it expects :



Consequently, irrespective of the number representation format (DEC Vax or IEEE standard), the solution obtained is very likely wrong, and may even vary from call to call because the "?" bits change even though the intended argument does not.

Bugs arising from mismatch of argument types occur quite frequently. Such bugs are rather difficult to diagnose during debugging on the DEC Vax because the results may be wrong to double-precision but look O.K. to single.

1.3.3 Underflow and Overflow Problems

We have discussed why the D format in DEC Vax is regarded as having too narrow an exponent field. If we use the same argument as for the Vax to decide how wide the range of the exponent field should be, we'll come to the conclusion that the exponent range for double precision in the IEEE standard is wider than necessary because we can raise the rounding error to a pretty high power before it underflows. However, there are reasons for using 8 and 11 bits for single and double precisions, respectively. The reason for using double

precision numbers is not merely for adding precision to a computation when there isn't enough precision in single precision numbers, but also to cope with overflow and underflow problems when the exponent range is inadequate.

Consider the evaluation of the polynomial

$$P(x) = a_0x^n + a_1x^{n-1} + \ldots + a_{n-1}x + a_n$$

where the coefficients of the polynomial are single precision numbers. We will discover fairly quickly that we'll encounter serious underflow and overflow problems even for modest values of n. For example, suppose $n = 7$, $x = 10^{10}$ and $\mid a_0 \mid \approx 1.0$. Although these values are not unreasonable, the value of $P(x)$ easily exceeds the range of single precision numbers. The largest single precision number allowed is 10^{38}, but $P(x)$ is of the order 10^{70}. So it is not uncommon to encounter overflow and underflow problems when we work in single precision for polynomials of reasonable degrees.

It turns out that if a polynomial has very large degree, we will encounter several other problems too. A majority of polynomials of large degrees suffer terribly from rounding errors. This is the reason for not evaluating $P(x)$ the obvious way, but using Horner's recurrence instead. If we have an eigenvalue problem where the matrix is 100×100, instead of evaluating its characteristic polynomial $P(x)$ of degree 100, we operate on the matrix in a very unobvious manner. We almost never compute $P(x)$ directly because to do so we need the coefficients of the polynomials. Rounding errors in the coefficients and in evaluating $P(x)$ can very often destroy the roots of the polynomial. So, besides the risk of overflow and underflow problems in the process of evaluating $P(x)$ directly, there is also the risk of inaccurate results arising from inaccurate coefficients.

It is not uncommon to evaluate polynomials of small degrees directly. Even though the effects of roundoff errors are not so severe, there is still the possibility of range problems. Double precision numbers can be used to circumvent this problem by evaluating and storing $P(x)$ as a double precision entity. By adding 3 bits to the exponent, we can raise the smallest or the largest number to the power of 8 before underflow or overflow problems occur. This is a generous increase in the range of permissible numbers.

1.4 Thou Shalt Have Single Precision Numbers

There is a peculiarity in the IEEE standard 754 which states that we must have single precision numbers, but double precision numbers are optional. Some implementations support just single precision numbers and not double precision numbers; this is often adequate for signal processing and graphics. Otherwise, most people compute in double precision most of the time, if they can.

Do they really need single precision numbers? By the rule of thumb argument discussed in lecture 2, single precision numbers are often not sufficient in most of our computations, but double precision numbers are often more than enough. This argument is unexpected and may not be easily understood by naive users, most of whom would argue that single precision numbers are adequate because most data can be stored in single precision and single precision numbers require less storage and shorter data transmission time. This argument happens not to be the chief reason for the need of single precision numbers.

The principal reason for using single precision numbers is that it is easier to debug numerical software when single precision numbers are used. It is extraordinarily difficult to debug software with a wide word because the probability of discovering that something has gone wrong is, roughly speaking, proportional to $2^{-(\text{number of bits})}$. Recall that $A = B \,\square\, C$ is

stored as $a = (b \square c)(1+\xi)$ in memory. The probability of detecting an error is proportional to ξ. Since $\xi \approx 2^{-23}$ in single precision and $\xi \approx 2^{-53}$ in double precision, it is much more likely for us to discover that something has gone wrong in the former. See J. Demmel's paper on "The Probability That A Numerical Analysis Problem Is Difficult", in *Mathematics of Computation*, April 1988.

1.4.1 Rounding Errors – Singularities and Unstable Algorithms

A problem can be mapped into an n-dimensional space where a point in the space is defined by the data of the problem. For instance, the quadratic equation $ax^2 + bx + c = 0$ can be mapped into a 3-dimensional space where a point in space is determined by the coefficients a, b and c. Similarly, the solution of a problem can be mapped into the solution space. The solution space for $ax^2 + bx + c = 0$ contains all possible roots of all quadratic equations. Consequently, solving a problem is essentially the process of mapping a point in the problem space to a point in the solution space.

For certain problems, surfaces exist in the problem space such that points lying on the surfaces represent singularities. If we attempt to solve a problem whose point lies on one of these surfaces, we will encounter problems. For instance, quadratic equations whose coefficients are very close to those with double roots often require many extra digits in the computations. These are equations where if we alter the coefficients slightly we obtain double roots.

Referring to Figure 1, for most points in the problem space, rounding errors of the order ε will result in errors proportional to $\frac{\varepsilon}{d}$ in the solution. The solution error can be worse if the point is near a self-intersecting surface of singularities, or it can be better in the sense that it can lose at most half the figures carried. The following example illustrates the latter :

Consider the matrix inversion problem

$$\text{data } A \longrightarrow \text{ solution } X = A^{-1}$$

where the data A is subject to an error ΔA inducing a corresponding error ΔX in X :

$$(A + \Delta A)^{-1} = X + \Delta X$$

and if

$$\varepsilon \approx \frac{\| \Delta A \|}{\| A \|}$$

then

$$\Delta X \approx -X(\Delta A)X$$

with relative error

$$\frac{\| \Delta X \|}{\| X \|} \approx \| \Delta A \| \cdot \| X \| .$$

The distance of A to the nearest singular matrix

$$d \approx \frac{1}{\| X \|},$$

and since

$$\frac{\| \Delta X \|}{\| X \|} \approx \frac{\| \Delta A \|}{d} \text{ and } \| \Delta A \| \approx \varepsilon \| A \|$$

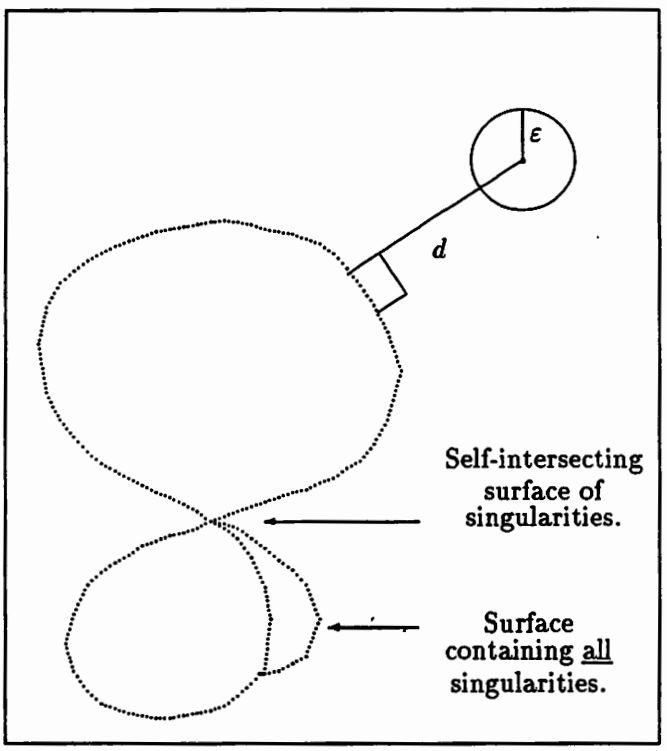


Figure 1: Data space and surface of singularities.

the relative error

$$\frac{\| \Delta X \|}{\| X \|} \approx \frac{\varepsilon \| A \|}{d}.$$

The number of figures carried is $-\log(\varepsilon)$ and the number of figures in agreement with a point on the surface is $\log(\frac{\|A\|}{d})$. The number of correct figures in X is

$$\begin{aligned} -\log(\frac{\| \Delta X \|}{\| X \|}) &= -\log(\varepsilon) - \log(\frac{\| A \|}{d}) \\ &= \text{numbers of figures carried} - \text{numbers of figures of agreement} \end{aligned}$$

Hence the numbers of figures lost equals the number of figures in agreement, which implies that at most half the figures carried are lost in this case.

The above discussion implies that the relative error is smaller in double precision than in single precision because ε is smaller in the former. Hence, if a point lies near a surface of singularities, we are more likely to detect that something has gone wrong if single precision is used.

Another reason rounding errors are more noticeable in single precision has to do with the way we format computer output. We usually print floating-point numbers up to approximately six decimal digits because this is about the number of digits an average person can grasp comfortably. Because we often format the output to print no more than half a dozen digits, we do not notice the loss of digits to the right of the sixth digit. Since the number of digits we print in single precision is about the same as the number of digits present, any problem resulting from rounding errors is very noticeable, but if double precision is used and the lowest few digits are perturbed, the perturbations are unobvious because some of the digits are not printed. Consequently, there's a higher probability that anomalies in results are detected when single precision is used.

Another problematic surface in the problem space is a result of the method used to solve a problem. For instance, the roots of quadratic equations whose roots are of great disparity - that is, one is tiny and the other is gargantum - are well defined. However if we use the formula $x = \frac{-b \pm \sqrt{(b^2-4ac)}}{2a}$ to determine the roots, we may obtain unexpected results. The coefficients of equations whose roots are of great disparity satisfy $b^2 \gg ac$. When $b^2 \gg ac$, by rounding off $4ac$, $b^2 - 4ac \approx b^2$ where the bottom few bits may be inaccurate. Consequently, the bottom few bits of $\sqrt{(b^2 - 4ac)} \approx b$ may also be wrong, and $b - \sqrt{(b^2 - 4ac)}$ may have a string of zeroes followed by these uncertain last few bits. When the result is normalized, it may appear perfectly all right, but it may actually be nonsense. Usually we do not realize that cancellation has occurred unless we follow the computations closely.

Without the above discussion, what is the probability that we'll encounter a problem which ultimately lead us to the conclusion that $x = \frac{-b \pm \sqrt{(b^2-4ac)}}{2a}$ is a bad formula? The probability of having massive cancellations in double precision is much lower than in single precision, so it is advisable to work in single precision during debugging because it enormously increases the odds, even when testing at random, of detecting symptoms of the loss in accuracy.

1.4.2 Debugging Strategies

Suppose we want to test our programs with a set of data, perhaps chosen at random. How do we know if our programs are producing the correct solutions? Well, we can run our

programs in both double and single precision and if their corresponding solutions disagree in early digits we know that something suspicious has happened.

Alternatively, we can substitute the solution obtained into the equation and see if both sides of the equation agree to a large extent. If substituting alleged roots for x makes the value of $ax^2 + bx + c$ very tiny or zero in magnitude, we know that our roots are quite reasonable. So, we really do not need double precision because by using the substitution technique with single precision we can actually determine if our roots are plausible by considering how well they satisfy the equation.

2 Simulating Single Precision with Double Precision

There are people who strongly believe that we should build only double precision hardware. With double precision, we can actually compute as if to single precision by computing each operation in double precision and then rounding it.

One of the advantages of implementing only double precision is that we can optimize the hardware for double precision. One such example is the SPUR project at Berkeley.

Let us use the following notation :

$$[expression] = (expression) \text{ rounded correctly to } p \text{ significant digits, and}$$
$$\square = \text{an operation like } +, -, \times \text{ or } /; \text{ not } \sqrt{}.$$

As usual the radix $\beta \in \{2, 8, 10 \text{ or } 16\}$.

Exercise : By how much must q exceed p to guarantee that

$$[[x \square y]_q]_p = [x \square y]_p$$

for all $x = [x]_p$ and $y = [y]_p$?

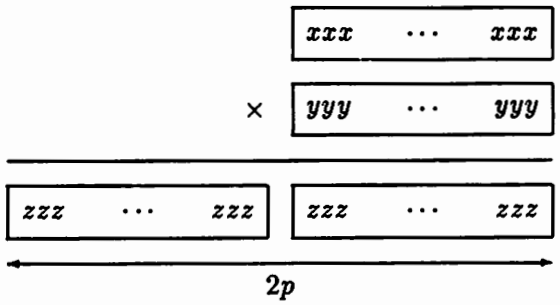
Answer : $q = 2p$ is enough if $\beta \geq 4$ or if $\square \neq \pm$; $q = 2p + 1$ is enough, always.

In the IEEE standard, the number of significant digits in the double precision is more than twice that of the single precision. Thus, in the standard we can simulate single precision arithmetic by double precision arithmetic provided we have the "Round-to-Single" operation follow each double-precision arithmetic opration. What follows are the proofs that the **Answer** is correct.

2.1 $\square$ is $\times$

Theorem 1 If $[x]_p = x$ and $[y]_p = y$ then $[x \times y]_{2p} = x \times y$.

Proof : When we multiply two integers the width of the product is at most the sum of the widths of the factors; to be precise, it is either the sum or the sum minus one because the leading digit may be zero. We can think of the numbers as if they are integers because the significant digits will not be altered and the point can be taken care of easily. The product of multiplying two p digits integers together is shown below :



INTEGERS

$$\beta^{p-1} \le y \le \beta^p - 1$$
$$\beta^{p-1} \le y \le \beta^p - 1$$
$$\beta^{2p-2} \le x \cdot y < \beta^{2p} - 1$$

From the discussion and the illustration above, it is obvious that $2p$ digits are sufficient. The examples below show that if you use fewer than $2p$ digits, you'll be sorry.

$$\begin{aligned} 100001 \times 111111 &= 100000\ 011111 \\ 111011 \times 101101 &= 100001\ 011111 \end{aligned}$$

$$\begin{aligned} 101111 \times 101111 &= 100010\ 100001 \\ 110001 \times 110001 &= 100101\ 100001 \end{aligned}$$

In the examples, $p = 6$ and $\beta = 2$. In this example, pairs of 6 bits numbers are multiplied and their results have the property that if they were first rounded to 11 bits and then to 6 bits, they would be incorrectly rounded.

The numbers of the example above were generated artfully with "P-Adic" arithmetic and Hensel's "Lifting". For examples with very small word-size, one can use exhaustive search. The artful technique will be discussed later when we discuss how to test multiplication and division.

2.2 □ is /

Theorem 2 If $[x]_p = x$ and $[y]_p = y$ then $[[x/y]_{2p}]_p = [x/y]_p$.

Proof : Normalize the quotient so that

$$1/\beta < q = x/y < 1,$$

and assume that y, the divisor, is an integer satisfying

$$\beta^{p-1} \le y \le \beta^p - 1.$$

The assumption on y can be accomplished by scaling. If $[q]_p = q$ then we are done, so let us assume that $[q]_p \neq q$. Let the two p-digit numbers adjacent to q be $\check{q} = [\check{q}]_p$ and $\hat{q} = [\hat{q}]_p$ where

$$\check{q} = \boxed{\cdot qqq \quad \cdots \quad qqq} < q < \hat{q} = \boxed{\cdot qqq \quad \cdots \quad qqq} + \boxed{\cdot 000 \quad \cdots \quad 001}$$

Let

$$\bar{q} = (\tfrac{q+\hat{q}}{2}) = \boxed{\cdot qqq \quad \cdots \quad qqq\delta}$$
$$\overleftrightarrow{\qquad\qquad} \; p+1$$

where $\delta = \frac{\beta}{2}$. Since we carry twice as many digits as p, we are done when $q = \bar{q}$. So, let us assume that $q \neq \bar{q}$ and

$$N = 2\beta^p \bar{q} = 2\times \boxed{qqq \quad \cdots \quad qqq \cdot \delta} \cdot$$

Multiplying $\bar{q}$ by β^p shifts the point immediately to the left of δ, and multiplying it by 2 makes N an integer (recall that $\delta = \frac{\beta}{2}$).

The magnitude of the difference between the quotient q and $\bar{q}$ is

$$|\, q - \bar{q} \,| = |\, \frac{x}{y} - \frac{N}{2\beta^p} \,| = |\, \frac{2\beta^p x - Ny}{2\beta^p y} \,|$$

which is a ratio of two integers because

$$\beta x = \beta q y > y \geq \beta^{p-1}.$$

Note that βx is as large as the smallest floating-point number beyond which all floating point-numbers are integers. The numerator $2\beta^p x - Ny$ is not zero because we assumed that $q \neq \bar{q}$, so we have

$$|\, q - \bar{q} \,| \geq \frac{1}{2\beta^p y} \geq \frac{1}{2\beta^p(\beta^p - 1)} = \frac{1}{2}\beta^{-2p}(1 + \beta^{-p} + \beta^{-2p} + \ldots)$$

which implies that $|\, q - \bar{q} \,| > \frac{1}{2}\beta^{-2p}$. Hence,

$$\text{either } q < \bar{q} - \tfrac{1}{2}\beta^{-2p} = \boxed{\cdot qqq \quad \cdots \quad qqq}\boxed{(\delta-1)\rho\rho\rho\cdots\rho\rho\rho}\,\delta$$

$$\text{or } q > \bar{q} + \tfrac{1}{2}\beta^{-2p} = \boxed{\cdot qqq \quad \cdots \quad qqq}\boxed{\delta 000 \quad \cdots \quad 000}\,\delta$$

When such a q is rounded to $2p$ digits and then to p digits, the results are the same as if it were rounded to p digits once. Therefore,

$$[[q]_{2p}]_p = [q]_p.$$

We have shown that $2p$ digits are sufficient for division, but are they necessary? Do we really need $q \geq 2p$ to ensure that

$$[[x/y]_q]_p = [x/y]_p?$$

The answer is *yes*! The following example demonstrates that $2p$ digits are necessary :

$$\begin{aligned} 5000/9999 &= 0.5000\ 5000\ \textit{5000}\ \textit{5000} \\ 4999/9999 &= 0.4999\ 4999\ \textit{4999}\ \textit{4999} \end{aligned}$$

If correctly rounded, the answers should be 0.5001 and 0.4999 respectively, but if less than $2p$ digits are used, they may both be 0.5000 instead. The above example is in decimal numbers; it is left as an exercise for interested readers to generate a similar example for binary numbers.

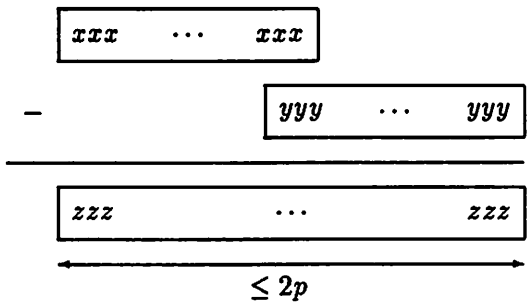
2.3 □ is -

Theorem 3 If $[x]_p = x > y = [y]_p > 0$ then $[[x-y]_{2p+1}]_p = [x-y]_p$.

Proof : Subtraction is the first disappointment because $2p+1$ digits are needed for rounding to work correctly. Actually $2p+1$ digits are needed in binary only; other radices require only $2p$ digits. We shall prove that $2p+1$ digits are sufficient and necessary by case analysis using pictures.

When we add two numbers, pre-shifting is often necessary to align the points of the operands. Consider the following cases :

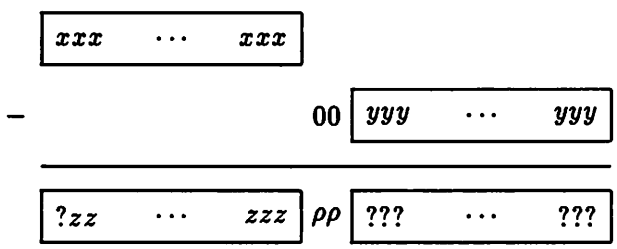
Case 1 : y is shifted $\leq p$ digits.



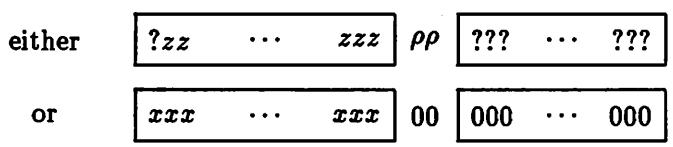
When y is shifted $\leq p$ digits, the result is no wider than $2p$ digits. Hence, $[x-y]_{2p} = x - y$.

Case 2 : y is shifted $\geq p+2$ digits.

Without loss of generality, assume that y is shifted by $p+2$ digits :



Recall that $\rho = \beta - 1$. When the result is rounded to $2p$ digits we may obtain



where no carry propagates out of the bottom p digits in the former but it does in the latter. When the former is subsequently rounded to p digits, it'll round up and produce x as the final solution, which is correct; the latter clearly rounds to x. So, $[x-y]_{2p} = x = [x-y]_p$.

Case 3 : y is shifted $p+1$ digits.

$$\begin{array}{rl} & \boxed{xxx \quad \cdots \quad xxx} \\ - & 0\,\boxed{yyy \quad \cdots \quad yyy} \\ \hline & \boxed{?zz \quad \cdots \quad zzz}\;\rho\;\boxed{zzz \quad \cdots \quad zzz} \end{array}$$

The most significant digit of the result may be a zero, in which case we have only $2p$ digits and there is no problem, that is, $[x-y]_{2p} = x - y$. Assume that the most significant bit is nonzero. When rounding to $2p$ digits and $\beta \neq 2$ we'll get

$$\boxed{zzz \quad \cdots \quad zzz}\;\rho\;\boxed{\text{non-zeroes}}$$

and consequently $[[x-y]_{2p}]_p = x = [x-y]_p$.
However, if $\beta = 2$ and we round to even, using $2p$ bits is not sufficient. Consider the following situation :

$$\begin{array}{rl} & \boxed{xxx \quad \cdots \quad xxx} \\ - & 0\,\boxed{111 \quad \cdots \quad 111} \\ \hline & \boxed{zzz \quad \cdots \quad zz\tilde{z}\;1\;000 \quad \cdots \quad 001} \\ & \longleftrightarrow \\ & 2p+1 \end{array}$$

When rounded to $2p$ digits we get

$$\boxed{zzz \quad \cdots \quad zz\tilde{z}\;1\;000 \quad \cdots \quad 000}$$

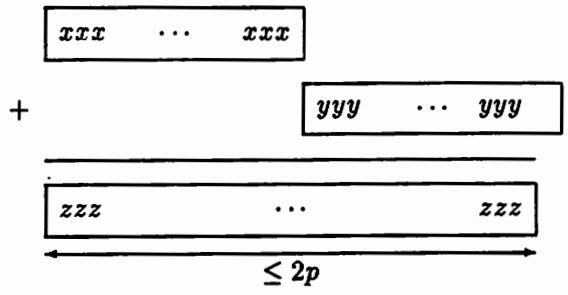
When the above is further rounded to even to p digits, the final answer depends on the digit $\tilde{z}$: if $\tilde{z} = 0$ then we round down; otherwise, we round up. Therefore, $[[x-y]_{2p}]_p < x = [x-y]_p$ when $\tilde{z} = 0$. In general, $[[x-y]_{2p+1}]_p = [x-y]_p$ holds for all radices.

2.4 $\square$ is $+$

Theorem 4 If $[x]_p = x \geq y = [y]_p > 0$ then $[[x+y]_{2p+1}]_p = [x+y]_p$.

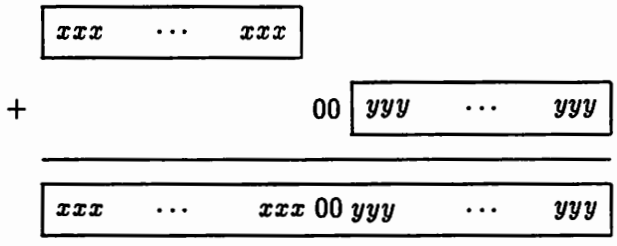
Proof : As in subtraction, $2p+1$ digits are needed for addition. Let us consider the following cases :

Case 1 : y is pre-shifted by $\leq p$ digits.

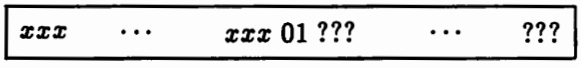


The result has no more than $2p$ digits, so $[x+y]_{2p} = x+y$. When y is preshifted by p bits, the most significant p digits of the result is x and y is the p least significant digits.

Case 2 : y is preshifted by $\geq p+2$ digits.

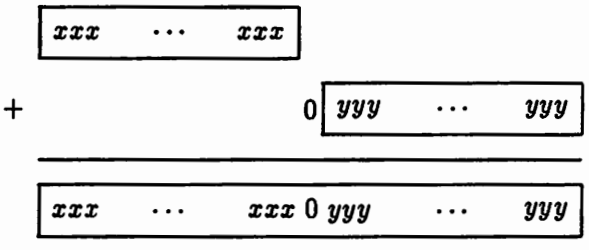


When rounded to $2p$ digits, we obtain either

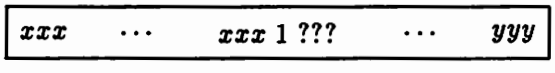


which is x when rounded to p digits. Hence, we have $[[x+y]_{2p}]_p = x = [x+y]_p$.

Case 3 : y is preshifted by $p+1$ digits.



when rounded to $2p$ digits, we may get



Using an argument similar to that of Case 3 for subtraction, we have

$$[[x+y]_{2p}]_p = x = [x+y]_p \quad \text{if} \quad \beta \geq 4$$
$$[x+y]_{2p+1} = x+y \quad \text{if} \quad \beta = 2$$

In the next lecture we shall discuss the case where $\square = \sqrt{}$. In summary, if we wish we may design our hardware optimized for double precision computations and achieve the effect of single precision computations by rounding down. In other words, we can convert single precision numbers to double precision, compute in double precision and then round to single precision as if single precision arithmetic were available. A consequence of this scheme is that single precision computations may be slowed down, relative to an independent single precision implementation; but because we speed up arithmetic generally, there are fewer arithmetic formats to decipher in the arithmetic unit, and double precision arithmetic may be faster.

Radix β $\in$ (2, 8, 10 or 16 ;

Precision p = no. of sig. digits of radix β

$[\text{expression}]_p$:= (expression) rounded correctly to p sig. dig.

$\circledast$:= operation $+, -, \times, /$ or $\surd$.

BY HOW MUCH MUST q EXCEED p

TO GUARANTEE THAT

$$[\,[x \circledast y]_q\,]_p = [x \circledast y]_p$$

FOR ALL $x = [x]_p$ AND $y = [y]_p$?

Answer: $q = 2p$ is enough if $\beta \geq 4$,
or if $\circledast$ is not $\pm$;

$q = 2p+1$ is enough, always.

PROOF: By PICTURES !

x = [xxx... xxx]

y = [yyy . yyy]

etc.

Application: (SPUR project, ...)

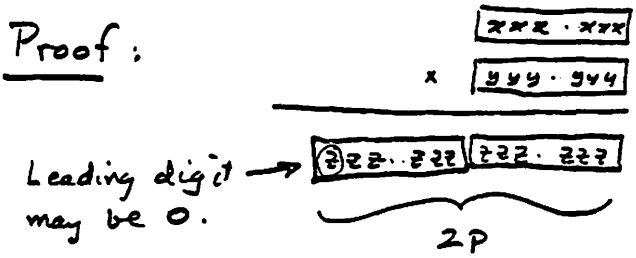
Hardware optimized for Double-Precision, with "Round-to-Single" operation

Kahan 26 May

$\otimes$ is $\times$:

> If $[x]_p = x$ and $[y]_p = y$
> then $[x \times y]_{2p} = x \times y$.

Proof :



INTEGERS

$\beta^{p-1} \le x \le \beta^p - 1$

$\beta^{p-1} \le y \le \beta^p - 1$

$\beta^{2p-2} \le x \cdot y < \beta^{2p} - 1$

Examples show any $q < 2p$ is not enough;

try $p = 6$, $\beta = 2$:

100001 x 111111 = 100000 011111 .

111011 x 101101 = 100001 011111

101111 x 101111 = 100010 100001

110001 x 110001 = 100101 100001

(Aside : how are examples like this generated?

1) Exhaustive search if p is small, or
2) "P-Adic" Arithmetic & Hensel "Lifting".)

⊛ is /:

> If $[x]_p = x$ and $[y]_p = y$
> then $[\,[x/y]_{2p}\,]_p = [x/y]_p$

Proof: Let $1/\beta < q := x/y < 1$,

and $\beta^{p-1} \le y \le \beta^p - 1$ ... y is an integer

If $[q]_p = q$ we are finished, so assume $[q]_p \neq q$.

Let $\check{q} = [\check{q}]_p$ and $\hat{q} = [\hat{q}]_p$ be adjacent p-sig.-digit numbers that straddle q;

$$\check{q} = .qqq\cdots qq? < q < \hat{q} = .qqq\cdots qqq + .000\cdots 001$$

Let $\bar{q} = (\check{q} + \hat{q})/2 = .qqq\cdots qqq\,\delta$ ($p-1$ digits) $\qquad \delta = \beta/2$

If $q = \bar{q}$ we are finished, so assume $q \neq \bar{q}$.

Now, $N = 2\beta^p \bar{q} = 2 \times .qqq\cdots qqq\cdot\delta$ is INTEGER

$$\therefore\ |q - \bar{q}| = \left|\frac{x}{y} - \frac{N}{2\beta^p}\right| = \left|\frac{2\beta^p x - Ny}{2\beta^p y}\right| = \frac{\text{integer}}{2\beta^p y}$$

since $\beta x = \beta q y \ge y \ge \beta^{p-1}$

$$\therefore\ |q - \bar{q}| \ge \frac{1}{2\beta^p y} \ge \frac{1}{2\beta^p(\beta^p - 1)} = \tfrac{1}{2}\beta^{-2p}(1 + \beta^{-p} + \beta^{-2p} + \cdots$$

$\therefore$ Either $q < \bar{q} - \tfrac{1}{2}\beta^{-2p} = .qqq\cdots qqq\,(\delta-1)ppp\cdots pp\,\delta$

or $q > \bar{q} + \tfrac{1}{2}\beta^{-2p} = .qqq\cdots qqq\,\delta\,000\cdots 00\,\delta$ ($2p$ digits)

$$\therefore\ [\,[q]_{2p}\,]_p = [q]_p .$$

Do we really need $q \geq 2p$

to ensure

$$[\,[x/y]_q\,]_p = [x/y]_p \quad ?$$

Try 4 sig. dec. :

$5000/9999 = 0.\underline{5000\ 5000}\ 5000\ 500\ldots$

$4999/9999 = 0.\underline{4999\ 4999}\ 4999\ 499\ldots$

Try 4 sig. bits :

$1000/1111 = 0.\underline{1000\ 1000}\ 1000\ 100\ldots$

$? = 0.1001\ 0111\ 0xxx\ldots$

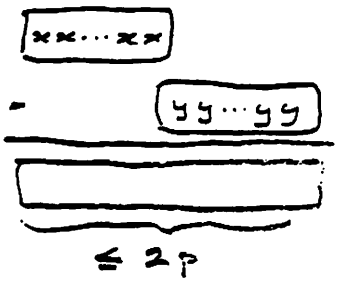
(*) is —

> If $[x_p] = x > y = [y_p] > 0$
> then $[\,[x-y]_{2p+1}]_p = [x-y]_p$

Proof. by cases, depending on pre-shift:

If y is shifted $\leq p$ digits

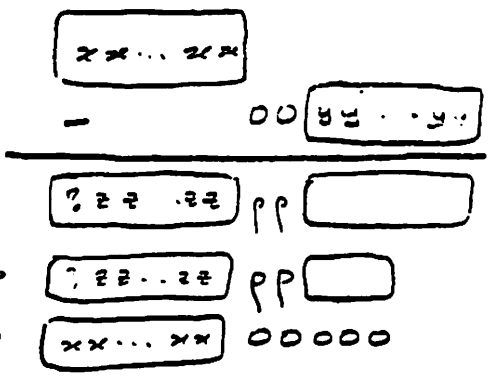
$$[x-y]_{2p} = x-y .$$



If y is shifted $\geq p+2$ digits

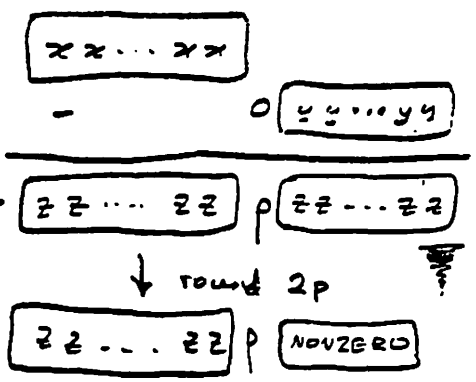
$$[x-y]_{2p} = x = [x-y]_p .$$

(Remember $\rho = \beta - 1$)



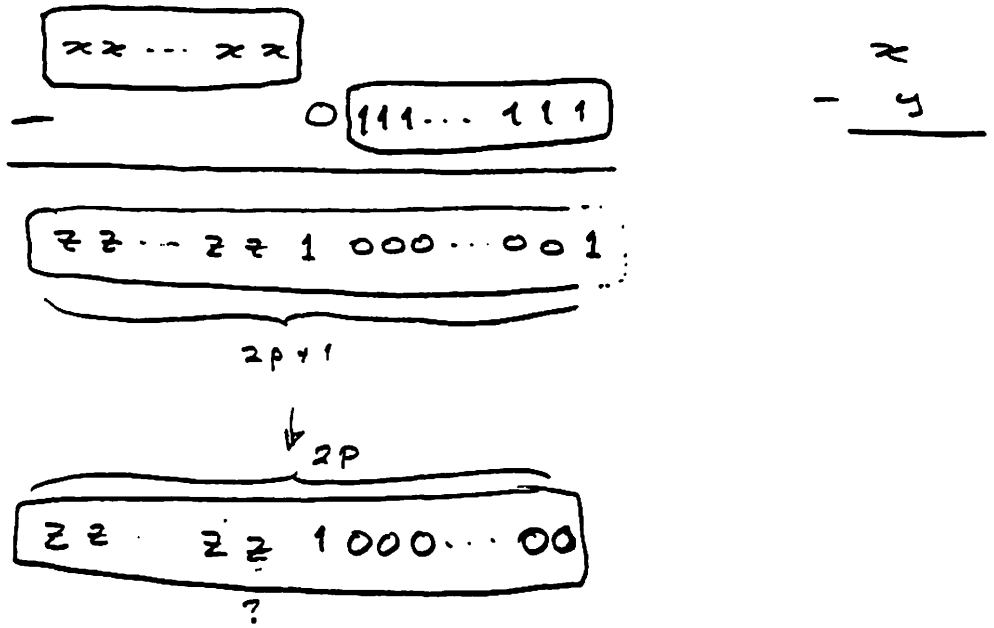
If y is shifted $p+1$ digits.

assume leading digit is nonzero, since otherwise $[x-y]_{2p} = x-y$



Then $[\,[x-y]_{2p}]_p = x = [x-y]_p$.

Exception: when $\beta = 2$ & round-to-even



$\therefore$ $[[x-y]_{2p}]_p < x = [x-y]_p$

in this case.

$\therefore$ In general $[[x-y]_{2p+1}]_p = [x-y]_p$

$\circledast$ is $+$:

> If $[x]_p = x \geq y = [y]_p > 0$
> then $[[x+y]_{2p+1}]_p = [x+y]_p$

Proof:

If y is preshifted by $< p$ digits,

$$\begin{array}{r} xx\cdots xx \\ +\ yy\cdots yy \\ \hline ?\,\square \end{array} \quad \underbrace{}_{\leq 2p}$$

or if y is preshifted by p digits,

$$\begin{array}{r} xx\cdots xx \\ +\ yy\cdots yy \\ \hline xx\cdots xx\,yy\cdots yy \end{array}$$

then $[x+y]_{2p} = x+y$.

If y is preshifted by $\geq p+2$ digits

$$\begin{array}{r} xx\cdots xx\ 00\ yy\cdots yy \\ \hline xx\cdots xx\,00\,yy\cdots yy \\ xx\cdots xx\,01\,??? \end{array} \quad \underbrace{}_{2p}$$

then $[[x+y]_{2p}]_p = x = [x+y]_p$.

If y is preshifted by $p+1$ digits ... $xx\cdots xx\,1\,???$

then

if $\beta \geq 4$, $[[x+y]_{2p}]_p = x = [x+y]_p$

if $\beta = 2$ $[x+y]_{2p+1} = x+y$.

$\circledast = \sqrt{\ }$:

It turns out that if $[x]_p = x > 0$

then $[\,[\sqrt{x}\,]_{2p}\,]_p = [\sqrt{x}\,]_p$.

We shall prove this (and more)

later.